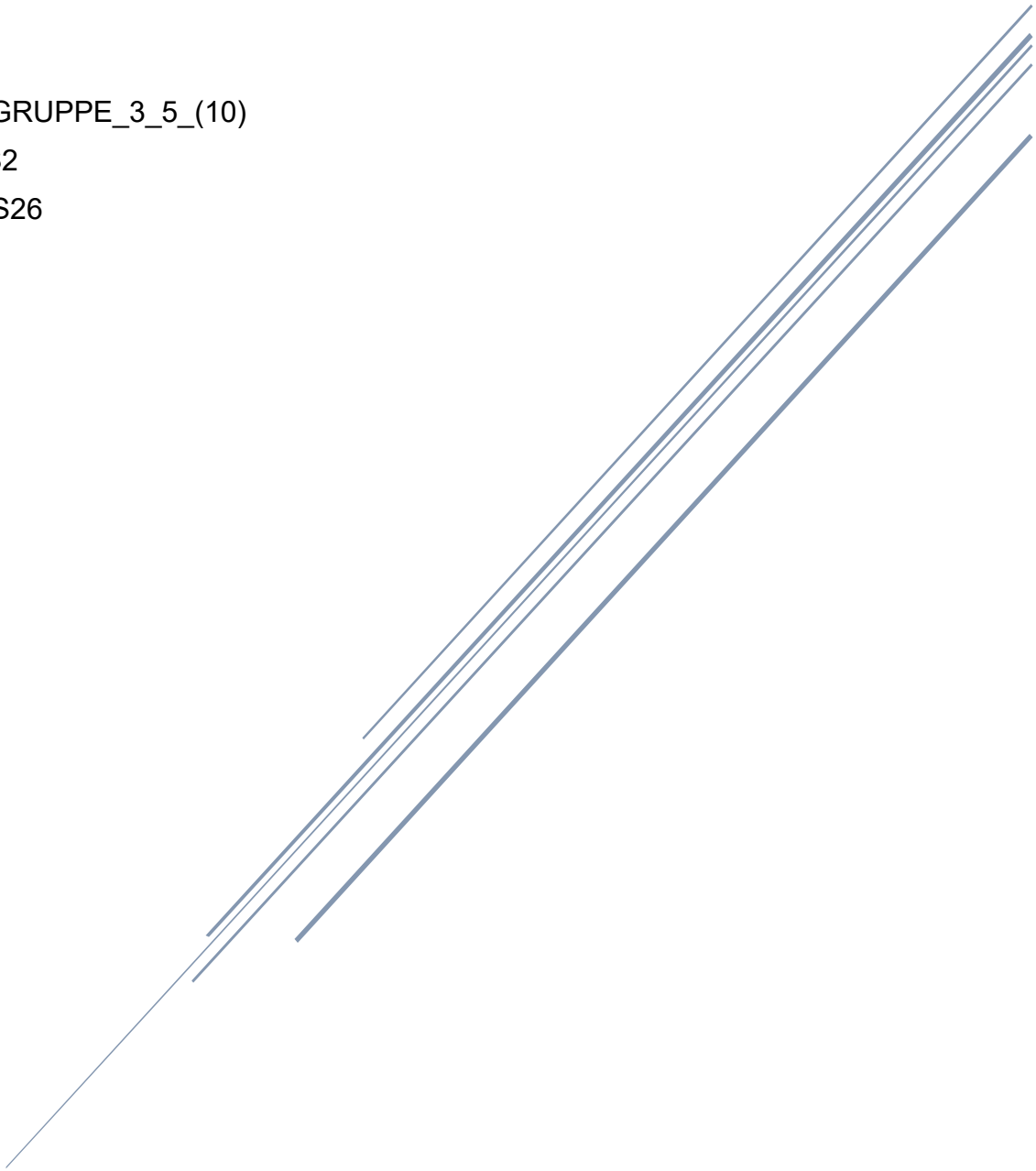
KÜNSTLICHE INTELLIGENZ DOKUMENTATION

Marc Pfitzenmaier (6839058), Björn Bauer (9973491)

Gruppe: GRUPPE_3_5_(10)

Projekt: S2

Daten: DS26



DHBW
Künstliche Intelligenz

Inhalt

1. Konzept

Im Zuge dieser Beschreibung werden an mehreren Stellen Codebeispiele gezeigt. Hierbei handelt es sich jeweils um Ausschnitte aus diesem Projekt. Es wird nicht jede einzelne Codezeile hier gezeigt oder in Ihrer Funktion beschrieben.

1.1. Annahmen für das Programm

- Der Spieler ist allwissend. Er sieht alle Spielsteine (das gesamte Labyrinth) und den Spielstein außerhalb.
- Das Bewegen der Figur ist optional, es kann demnach mehrmals hintereinander das Labyrinth verschoben werden.
- Die Position des Einschubs des nicht genutzten Spielsteins wird durch den Spieler bestimmt. Der Spielstein kann sich nicht drehen.
- Die Spielfigur wird, wenn sie rausfliegt auf das Feld gesetzt, dass sie hinausgeschoben hat (das Feld, das hineingeschoben wurde).
- Der Spieler startet auf keinem Feld in dem 5*4 Felder großen Labyrinth. Das Ziel liegt ebenfalls außerhalb. (Er startet eine Reihe unter der letzen Reihe, die zu dem Labyrinth gehört und das Ziel befindet sich eine Reihe über der ersten Reihe, die zum Labyrinth gehört)

1.2. Einlesen der Daten (Labyrinth, Start, Ziel, freies Feld)

Bei Start des Programms müssen die oben genannten Daten eingelesen werden. Das Format dabei ist:

- Labyrinth
  - eine zweidimensionale Arraylist in der 5*4 Felder beschrieben sind (separiert durch, in Form eines Arrays). Art des Feldes durch Code angegeben.
  - Zusätzlich werden zwei weitere Reihen eingefügt, um den Start und das Ende zu symbolisieren.
- Start/Ziel
  - *„startingPosition": { "row": 6, "column": 1 }*
  - *„goalPosition": { "row": 0, "column": 2 }*
- Freies Feld
  - *"freeTile": 2*
- Positionen bei denen das freie Feld eingefügt werden kann
  - *"rowsWithTileInput": { "left": [1, 3, 5], "right": [2, 4] }*

1.3.Labyrinth Felder

Es handelt sich um ein 5*4 Felder großes Labyrinth, das aus 10 (+2) verschiedenen Bausteinen bestehen kann. Diese Bausteine werden, wie Bild 1 zeigt codiert. Der Zahlencode bestimmt von welchen Feldern Sie erreichbar sind.

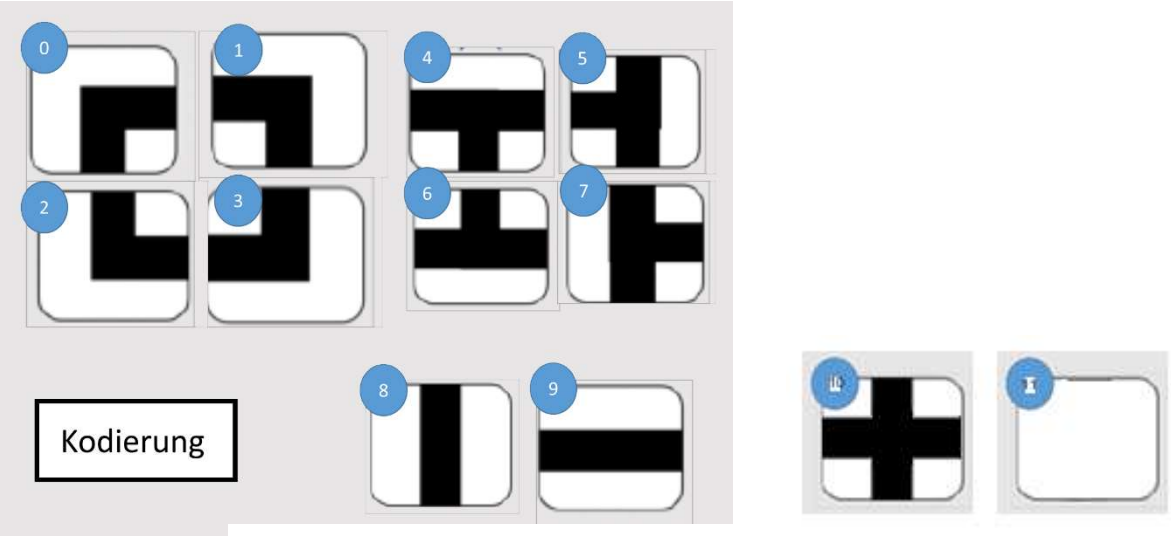


Abbildung 1: Codierung Labyrinth Bausteine

Jedes Labyrinth-Teil benötigt zusätzlich Regeln, in welche Richtung gegangen werden kann und welches Bauteil an dieser Stelle möglich ist. Ein Beispiel hierfür wäre das bei Bauteil 0 nur nach rechts gegangen werden kann und auch nur, wenn sich hier das Bauteil 1, 4, 5, 6, 3 oder 9 befindet. Außerdem kann nur nach unten gegangen werden, wenn sich hier Bauteil 2, 3, 8, 6, 5 oder 7 befindet. Ähnliche Regeln müssen für alle 10 Bauteile aufgestellt werden. Das untenstehende Codebeispiel zeigt dies in Bezug auf dieses Projekt.

Code:

```
class LabyrinthTile:
    """
    defines from which directions the labyrinth tile is accessible
    """

        def __init__(self, left_access: bool, top_access: bool, right_access: bool, bottom_access:
        bool):
    """
        Creates a labyrinth tile and defines from which direction it is accessible
        Args:
            left_access: defines if the tile can be accessed from the left
            top_access: defines if the tile can be accessed from the top
            right_access: defines if the tile can be accessed from the right
            bottom_access: defines if the tile can be accessed from the bottom
```

```
    """

        self.left_accessible = left_access
        self.top_accessible = top_access
        self.right_accessible = right_access
        self.bottom_accessible = bottom_access

    def __eq__(self, other):
        if not isinstance(other, LabyrinthTile):
            # don't attempt to compare against unrelated types
            return NotImplemented

        return self.left_accessible == other.left_accessible and self.top_accessible ==
other.top_accessible and self.right_accessible == other.right_accessible and
self.bottom_accessible == other.bottom_accessible

-----------------------------------------------------------------------------------------------------------------------

LABYRINTH_FILE_DICTIONARY =
{
                                        "0": LabyrinthTile(False, False, True, True),
                                        "1": LabyrinthTile(True, False, False, True),
                                        "2": LabyrinthTile(False, True, True, False),
                                        "3": LabyrinthTile(True, True, False, False),
                                        "4": LabyrinthTile(True, False, True, True),
                                        "5": LabyrinthTile(True, True, False, True),
                                        "6": LabyrinthTile(True, True, True, False),
                                        "7": LabyrinthTile(False, True, True, True),
                                        "8": LabyrinthTile(False, True, False, True),
                                        "9": LabyrinthTile(True, False, True, False),
                                        "10": LabyrinthTile(True, True, True, True),
                                        "11": LabyrinthTile(False, False, False, False)
                                    }

    """
            Maps LabyrinthTile code to correct LabyrinthTile object
            - First Boolean Parameter: Access to the left side of the tile
            - Second Boolean Parameter: Access to the top side of the tile
            - Third Boolean Parameter: Access to the right side of the tile
            - Fourth Boolean Parameter: Access to the bottom side of the tile
              For Exam
    """
```

Zu beachten hierbei sind die Bausteine 10 und 11. Beide befinden sich nur in den bereits erwähnten, zusätzlichen Zeilen und Symbolisieren den Start und das Ziel. Bauteil 10 hat eine Verbindung zu allen Seiten (Ein Kreuz), Bauteil 11 ist ein Wand-Bauteil, hier gibt es keine Möglichkeit ein anderes Bauteil zu erreichen. Beide Zeilen stehen für das Einfügen von nicht genutzten Labyrinth Bausteinen nicht zur Verfügung.

1.4.Verschieben Spielfigur (A*)

Die Spielfigur kann nur verschoben werden, wenn es einen Weg gibt auf dem Sie Laufen kann (der Weg wird in den Abbildungen Schwarz dargestellt). Dies betrifft auch Start und Ziel.
Die Implementierung dieses Verhalten kann in der Datei Labyrinth.py in der Funktion „def move_player(self)“ näher betrachtet werden.

1.5.Verschieben Labyrinth

Das Labyrinth kann zu jeder Zeit verschoben werden, es ist nicht nötig, dass sich die Spielfigur zwischen dem Verschieben von Bauteilen bewegt. Spielsteine können nur an fünf spezifizierten Punkten eingesetzt werden (in Abbildung 2 durch blaue Pfeile markiert). Wie bereits erwähnt kann in den Zusatzreihen kein Bauteil eingefügt werden. Die hierzu wichtigen Elemente im Code sind die Funktionen

„def insert_free_tile(self, left_rows: List[int], right_rows: List[int])*” (Labyrinth.py)*

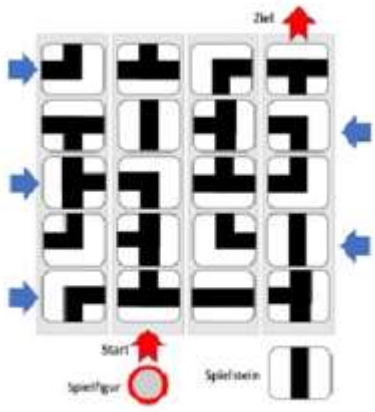


Abbildung 2: Labyrinth Grundlage

1.6.Spielzug

Ein Spielzug besteht aus dem Einsetzen eines Bauteils und dem anschließenden Bewegen der Spielfigur. Die Spielfigur muss sich allerdings nicht bewegen, weshalb jedes Einsetzen eines Spielsteins als einzelner Spielzug anzusehen ist. (Ein Spielzug kann aus mehreren Steps, also dem Einfügen eines Bauteils und der Bewegung der Spielfigur bestehen, muss dies aber nicht)

1.7.A* Implementierung

An dieser Stelle soll nicht erklärt werden, was unter dem A*-Algorithmus verstanden wird oder wie dieser Algorithmus angewandt wird. Es geht hier nur darum zu verdeutlichen, welche Elemente des Codes die Implementierung des Algorithmus in diesem Projekt darstellen. Der Hauptteil der Implementierung des A* befindet sich in der Datei AStar.py. Zusätzlich muss für diese Heuristische Suchverfahren die Heuristik berechnet werden. Die Hierzu notwendige Funktion finden Sie in der Datei Labyrinth.py (*def calculate_combined_heuristic_cost(self) -> float).* Kosten setzen sich dabei aus dem Einschieben eines Spielsteins (1) und der Bewegung der Spielfigur (0) zusammen. Näheres zur Berechnung der Heuristik finden Sie unter Punkt 3. Begründung des Entwurfs/der Umsetzung (Diskutieren der Konfiguration).

1.8.Spielfigur neu setzen

Wird die Spielfigur aus dem Spielfeld geschoben (durch das Einsetzen eines Spielsteins) so wird Sie auf den Spielstein, welcher hineingeschoben wurde, wieder eingesetzt. Dieses Verhalten wird in Bild 3 verdeutlicht. Dieses Verhalten wird in derselben Funktion, in der ein neuer Baustein eingefügt wird, implementiert. (*def insert_free_tile(self, left_rows: List[int], right_rows: List[int]*) Labyrinth.py)

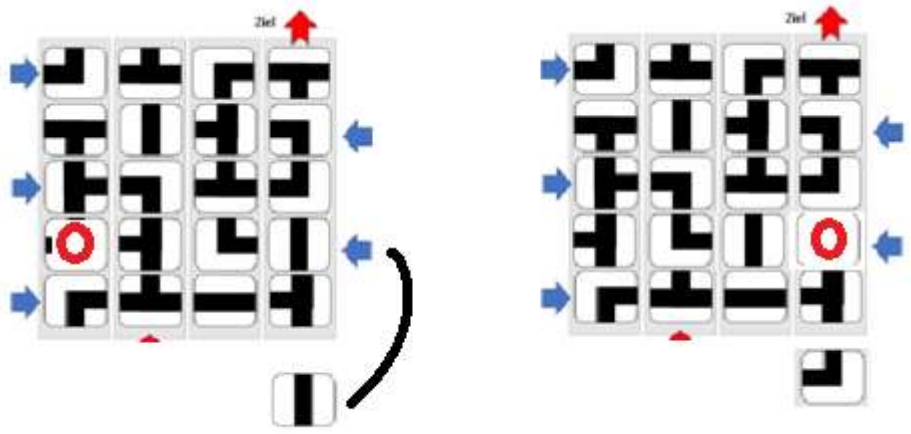


Abbildung 3: Wiedereinfügen einer herausgeschobenen Spielfigur

2. Bezug zur Vorlesung

Bei dieser Aufgabe handelt es sich um die Implementierung des A*-Algorithmus (siehe Abbildung 4), ein heuristisches Suchverfahren (Vorlesung Intelligente Suchverfahren in der Veranstaltung Künstliche Intelligenz) zur Bestimmung des kürzesten Wegs von einem Starpunkt zu einem Zielpunkt. Besonderheit bei der Problemstellung dieses Projektes ist, dass grundsätzlich zwei A* nötig sind, einen der den besten Weg für die Spielfigur findet und nach jedem Einsetzen des Spielsteins neu gestartet werden muss und einer, der den besten Punkt für das Einsetzen des Spielsteins bestimmt.

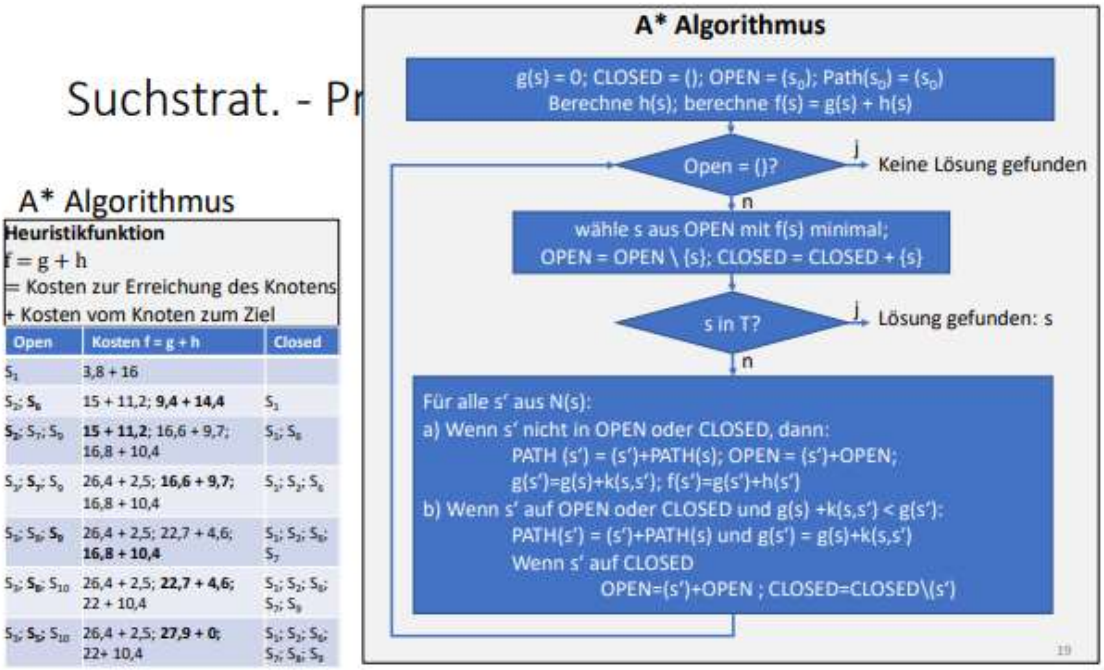


Abbildung 4: A*-Algorithmus

Eine der Besonderheiten dieses Algorithmus ist, dass bereits verworfene Wege unter bestimmten Umständen wieder verwendet werden können. Zu Sehen ist dieses Verhalten in Abbildung 4 unter Regel b) „Wenn s‘ auf closed“.

Wie bereits angesprochen handelt es sich bei diesem Algorithmus um ein heuristisches Suchverfahren. In diesem Zusammenhang bezieht sich der Begriff der Heuristik auf eine Technik, die entwickelt wurde, um ein Problem schneller oder effektiver zu lösen, insofern klassische Methoden zu langsam/ineffektiv sind.

3. Begründung des Entwurfs/der Umsetzung (Diskutieren der Konfiguration)

Da eine detaillierte Beschreibung jeder Einzelheit der Umsetzung den Rahmen dieser Dokumentation sprengen würde, werden hier nur einige Elemente genauer erörtert. Dies betrifft die Wahl der Programmiersprache, die Implementierung des Labyrinths selbst, die Implementierung des A*-Algorithmus die gewählte Heuristik, Input/Output und die Einführung neuer Elemente.

Die Programmiersprache

Für dieses Projekt ist die Programmiersprach der Wahl Python, eine Sprache, die in den Bereichen KI vermehrt zum Einsatz kommt. Neben dieser Tatsache sind mögliche mitgelieferte Funktionalitäten, die das Projekt erleichtern könnten, der Hauptgrund für den Einsatz der Sprache.

Das Labyrinth

Das Labyrinth wird durch ein zweidimensionales Array implementiert. Dies ist nach unserer Auffassung die einfachste Implementierung für ein Labyrinth dieser Größe. Nicht nur können alle Anforderungen, welche an dieses Labyrinth gestellt werden, erfüllt werden, auch eine Skalierung nach oben und unten ist ohne weiteres möglich. (Sofern weiterhin jede Spalte und jede Reihe die Gleiche Anzahl an Elementen hat. Auch die Ausführgeschwindigkeit leidet nach unseren Tests nicht durch diese Art der Implementierung. (Es kann keine Aussage getroffen werden, inwiefern die Größe des Labyrinths hierzu beiträgt, da Tests mit kleinen Labyrinthen gemacht wurden)

Da es in Python speziell mehrere Implementationen eines Arrays gibt, wurden diesbezüglich zwischen den Numpy-Arrays und den Python Arrays Tests durchgeführt, um bei der Wahl der Arrays eine ideale Performance zu erhalten. Diese Tests finden Sie hier: KI/Other/TestingNumpyVsPythonArrays.ipynb (Jupyternotebook zur besseren und leichteren Visualisierung)

Der A*-Algorithmus

Bei der Implementierung des A*-Algorithmus gibt es keinerlei Abweichungen zu einer Standard Implementierung. Im Gegensatz zu vielen anderen Implementationen des Algorithmus umfasst unsere Implementierung nicht nur einen Zustand, die Position und Bewegungsmöglichkeiten der Spielfigur, sondern auch den Zustand des Labyrinths welcher sich verändern kann. Trotz dieser Besonderheit entspricht die Implementierung dem in Abbildung gezeigten Vorgehen.

Die Heuristik

Die von uns gewählte Heuristik, um zu beurteilen, welcher Weg der Bessere ist bezieht sich auf die folgenden von uns festgelegten Regeln:

- Approximates the heuristic which at least is the distance + 1 (at least one tile insert is needed)
- also factors in the amount of tiles that have to be passed at a minimum
- passing a single tile has an approximated cost of less than one, because only tile insertions increase cost and multiple tiles can be passed per insertion. (0.3 arbitrary number)
- also rewards being positioned on a tile which has access in the directions towards the goal

Regel 1 ist selbsterklärend, jedes Feld hat eine gewisse Distanz zum Ziel und da das Labyrinth noch nicht als gelöst betrachtet werden kann muss bei jedem Weg + 1 gerechnet werden, da mindestens ein Baustein eingefügt werden muss. An dieser Stelle ist es unmöglich von einer +1 abzuweichen, da nach jedem Einfügen die Gesamtsituation neu begutachtet werden kann. Es ist nicht möglich in unserer Implementierung weiter in die Zukunft zu sehen.

Regel 2 ist ebenfalls selbsterklärend, je näher ein Feld am Ausgang ist desto besser ist das für die Kosten des Wegs. Dies soll allerdings nicht bedeuten, das nur Felder nahe des Ausgangs Beachtung finden, lediglich das diese Bevorzugt werden.

Regel 3 ist ein Kernelement unserer Heuristik, da hier ein Unterschied gemacht wird zwischen dem Einfügen eines Bausteins und dem Bewegen der Spielfigur. Es muss „teurer“ sein einen neuen Baustein einzufügen als den Spielstein zu bewegen. Dieser Zusammenhang ergibt sich aus der Tatsache, dass in einem Spielzug nur ein Baustein eingefügt werden kann, die Spielfigur sich aber eine unbegrenzte Anzahl an Feldern bewegen kann.
Der von uns hier angesetzte Wert von 0.3 für die Bewegung der Spielfigur ist ein durch Testen gefundener Wert. Nach unseren Tests muss dieser Wert unter 0.5 sein, kann demnach in der Theorie auch kleiner sein als 0.3. (Die einzige Bedingung, die hierfür unerlässlich ist, ist dass dieser Wert über 0 sein muss. Empfehlenswert ist nach unserer Einschätzung 0.3 – 0.5- Bei einem Wert über 0.5 werden durchschnittlich 2 Felder pro Zug durchschritten.)

Regel 4 schließt sich den Regeln 1 und 2 an. Da es bei dieser Aufgabe um den Versuch handelt, den kürzesten Weg in einem Labyrinth zu finden, muss ein Zugang in Richtung Ziel bevorzugt werden, denn so können mögliche Umwege, die zwar ebenfalls zu Ziel führen, aber nicht dem kürzesten Weg entsprechen vermieden werden.

Input/Output

Der Input bei diesem Projekt entspricht grundsätzlich den Vorgaben mit wenigen Ergänzungen. Diese Ergänzungen wurden gewählt, um uns den Zugang zu diesen Daten zu erleichtern und die Flexibilität im Projekt zu erhöhen. In der Theorie ist es möglich auch ohne diese kleinen Veränderungen die Eingabe zu nutzen. (Dies hängt allerdings von der Interpretation der Aufgabe ab. Ein Beispiel hierfür ist, ob der Spielstein bereits existiert und vor dem Labyrinth wartet oder erst wenn er das erste Feld erreichen kann gesetzt wird.)

Bei der Ausgabe handelt es sich um eine einfache, textbasierte Ausgabe. Nach unserer Auffassung ist das die übersichtlichste Möglichkeit der Ausgabe, ohne das Projekt um einen großen Teil (graphische Ausgabe) zu erweitern. Eine einfachere, graphische Ausgabe in Form eines Arrays oder verschiedener Linien wurde in Betracht gezogen aber aufgrund von Komplexität für den Leser der Ausgabe oder der Unübersichtlichkeit wieder verworfen.

Neue Elemente

Aufgrund unseres Verständnisses, dass die Spielfigur bereits existiert und vor dem Labyrinth „wartet“, haben wir uns dafür entschieden neue Elemente in Form zweier, weiterer Zeilen zum Input hinzuzufügen. Es wäre theoretisch möglich für dies Reihen bereits vorhandene Bausteine zu wählen, da beide Reihen nicht durch neue Bausteine verschoben werden können. Um klar darzustellen, dass es sich bei diesen Reihen um neue Elemente handelt haben wir uns allerdings dafür entschieden, auch neue, bisher nicht verwendete Elemente zu nutzen (ein leeres Element und ein Kreuz).

4. Test und Ergebnisbewertung (Diskutieren des Ergebnisses)

Das Labyrinth kann in 3 Zügen beendet werden (wie in Abbildung 5 gezeigt wird). Dabei muss vor jedem Laufen der Spielfigur ein Spielstein eingesetzt werden.

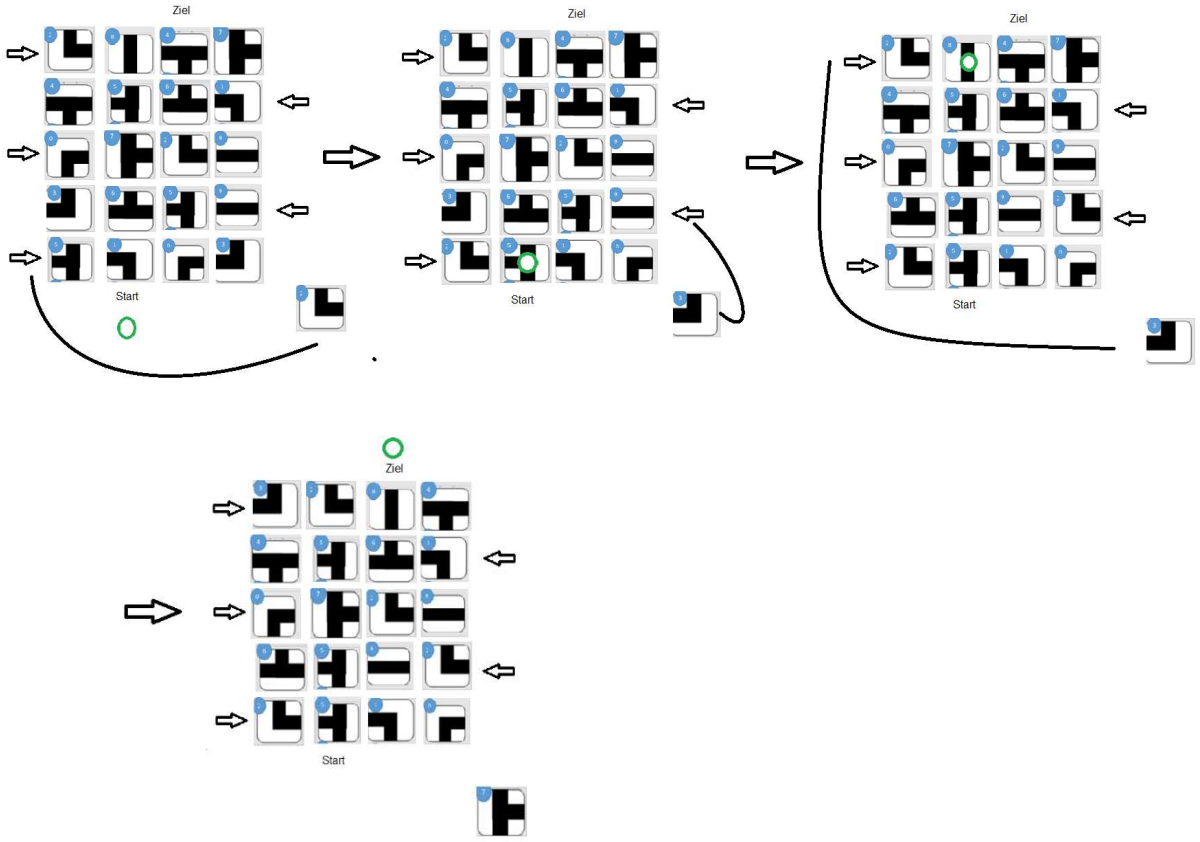


Abbildung 5: Ideale Lösen des Aufgaben-Labyrinth per Hand

Bei einem Test unserer Anwendung ergibt die Ausgabe folgendes:

The goal is row 0 and column 2
The starting position is row 6 and column 1
Step 1: The player inserted a tile at row 5 on the left side
Step 2: The player moved from row 6 and column 1 to row 5 and column 1
Step 3: The player inserted a tile at row 4 on the right side
Step 4: The player moved from row 5 and column 1 to row 1 and column 1
Step 5: The player inserted a tile at row 1 on the left side
Step 6: The player moved from row 1 and column 2 to row 0 and column 2
Using the A algorithm, the fasted way to reach the goal destination is 3 moves*

Da beide Lösungen (mit der Ausnahme, dass wir auf dem Papier keine weiteren Reihen benötigen, um den Start und das Ziel zu beschreiben) identisch sind, ist anzunehmen, dass das Programm für dieses Labyrinth perfekt arbeitet. Leider ist es schwer zu Beweisen, ob der Algorithmus auch für weitere Labyrinth auf diesem Level arbeitet, zumindest bei den von uns durchgeführten Test konnten wir keine Fehler feststellen. Eine Möglichkeit das Verhalten genauer zu Untersuchen wäre es zum Beispiel, das Labyrinth durch andere Algorithmen lösen zu lassen und die Ergebnisse zu vergleichen. Eine weitere wäre es verschiedene Implementierungen des A*-Algorithmus zu vergleichen, im Idealfall ist bei dieser Gruppe eine Implementierung dabei von der bekannt ist, wie gut diese Funktioniert und auch hier die Ergebnisse zu überprüfen.
Abschließend muss gesagt werden, dass es uns nur begrenzt möglich ist, eine Aussage über die Güte unserer Implementierung zu treffen.